LEAK-RESISTANT CRYPTOGRAPHIC METHOD AND APPARATUS

Inventors:
Paul C. Kocher
Joshua M. Jaffe

FIELD OF THE INVENTION

The method and apparatus of the present invention relate generally to cryptographic systems and, more specifically, to securing cryptographic tokens that must maintain the security of secret information in hostile environments.

BACKGROUND OF THE INVENTION

Most cryptosystems require secure key management. In public-key based security systems, private keys must be protected so that attackers cannot use the keys to forge digital signatures, modify data, or decrypt sensitive information. Systems employing symmetric cryptography similarly require that keys be kept secret. Well-designed cryptographic algorithms and protocols should prevent attackers who eavesdrop on communications from breaking systems. However, cryptographic algorithms and protocols traditionally require that tamper-resistant hardware or other implementation-specific measures prevent attackers from accessing or finding the keys.

If the cryptosystem designer can safely assume that the key management system is completely tamper-proof and will not reveal any information relating to the keys except via the messages and operations defined in the protocol, then previously known cryptographic techniques are often sufficient for good security. It is currently extremely difficult, however, to make hardware key management systems that provide good security, particularly in low-cost unshielded cryptographic devices for use in applications where attackers will have physical control over the

device. For example, cryptographic tokens (such as smartcards used in electronic cash and copy protection schemes) must protect their keys even in potentially hostile environments. (A token is a device that contains or manipulates cryptographic keys that need to be protected from attackers. Forms in which tokens may be manufactured include, without limitation, smartcards, specialized encryption and key management devices, secure telephones, secure picture phones, secure web servers, consumer electronics devices using cryptography, secure microprocessors, and other tamper-resistant cryptographic systems.)

A variety of physical techniques for protecting cryptographic devices are known, including enclosing key management systems in physically durable enclosures, coating integrated circuits with special coatings that destroy the chip when removed, and wrapping devices with fine wires that detect tampering. However, these approaches are expensive, difficult to use in single-chip solutions (such as smartcards), and difficult to evaluate since there is no mathematical basis for their security. Physical tamper resistance techniques are also ineffective against some attacks. For example, recent work by Cryptography Research has shown that attackers can non-invasively extract secret keys using careful measurement and analysis of many devices' power consumption. Analysis of timing measurements or electromagnetic radiation can also be used to find secret keys.

Some techniques for hindering external monitoring of cryptographic secrets are known, such as using power supplies with large capacitors to mask fluctuations in power consumption, enclosing devices in well-shielded cases to prevent electromagnetic radiation, message blinding to prevent timing attacks, and buffering of inputs/outputs to prevent signals from leaking out on I/O lines. Shielding, introduction of noise, and other such countermeasures are often, however, of limited value, since skilled attackers can still find keys by amplifying signals and filtering out noise by averaging data collected from many operations. Further, in smartcards and other tamper-resistant chips, these countermeasures are often inapplicable or insufficient due to reliance on external power sources, impracticality of shielding, and other physical constraints. The use of blinding and constant-time mathematical algorithms to prevent timing attacks is also known, but does not prevent more complex attacks such as power consumption analysis (particularly if the system designer cannot perfectly predict what information will be available to an attacker, as is often the case before a device has been physically manufactured and characterized).

The present invention makes use of previously-known cryptographic primitives and operations. For example: U.S. patent 5,136,646 to Haber et al. and the pseudorandom number generator used in the RSAREF cryptographic library use repeated application of hash functions; anonymous digital cash schemes use blinding techniques; zero knowledge protocols use hash functions to mask information; and key splitting and threshold schemes store secrets in multiple parts.

SUMMARY OF THE INVENTION

The present invention introduces leak-proof and leak-resistant cryptography, mathematical approaches to tamper resistance that support many existing cryptographic primitives, are inexpensive, can be implemented on existing hardware, and can solve problems involving secrets leaking out of cryptographic devices. Rather than assuming that physical devices will provide perfect security, leak-proof cryptographic systems are designed to remain secure even if attackers are able to gather some information about the system and its secrets. One of the essential attributes of a leak-proof cryptosystem is that it is "self-healing" such that the value of information leaked to an attacker decreases or vanishes with time. This invention describes leak-proof and leak-resistant systems that implement symmetric authentication, Diffie-Hellman exponential key agreement, ElGamal public key encryption, ElGamal signatures, and the Digital Signature Standard.

Leak-proof cryptosystems are able to withstand leaks of up to L_{MAX} bits of information per transaction, where L_{MAX} is a security factor chosen by the system designer to exceed to the maximum anticipated leak rate. The more general class of leak-resistant cryptosystems encompasses those that can withstand leaks, but are not necessarily defined to withstand any defined maximum information leakage rate. Both the leak-proof and leak-resistant systems of the present invention can survive a variety of monitoring and eavesdropping attacks that would break traditional (non-leak-resistant) cryptosystems.

A typical leak-resistant cryptosystem of the present invention consists of three general parts. The initialization or key generation step produces secure keying material appropriate for the

scheme. The update process cryptographically modifies the secret key material in a manner designed to render useless any information about the secrets that may have previously leaked from the system, thus providing security advantages over systems of the background art. The final process performs cryptographic operations, such as producing digital signatures or decrypting messages.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 shows a leak-proof symmetric authentication method.

Figure 2 shows a leak-resistant Diffie-Hellman exponential key exchange operation.

Figure 3 shows a leak-resistant RSA private key operation.

Figure 4 shows a leak-resistant ElGamal signing operation.

DETAILED DESCRIPTION OF THE INVENTION

Introduction to Symmetric Leak-Proof Cryptography

The leakage rate L is defined as the number of bits of useful information about a cryptosystem's secrets that are revealed per operation, where an operation is a cryptographic transaction. Although an attacker may be able to collect more than L bits worth of measurement data, by definition this data yields no more than L bits of useful information about the system's secrets.

The implementer of a leak-resistant system must choose a design parameter L_{MAX}, the maximum amount of leakage per operation the system may allow if it is to remain uncompromised. L_{MAX} should be chosen conservatively, and normally should significantly exceed the amount of useful information known to be leaked to attackers about the system's secrets during each transaction. Designers do not necessarily need to know accurately or completely the quantity and type of information that may leak from their systems; the choice of L_{MAX} may be

made using estimates and models for the system's behavior. General factors affecting the choice of L_{MAX} include the types of monitoring potentially available to attackers, the amount of error in attackers' measurements, and engineering constraints that limit L_{MAX}. (Larger values of L_{MAX} increase memory and performance requirements of the device, and in some cases may increase L.) To estimate the amount of useful information an attacker could collect by monitoring a device's power consumption, for example, a designer might consider the amount of noise in the device's power usage, the power line capacitance, the useful time resolution for power consumption measurements, as well as the strength of the signals being monitored. Similarly, the designer knows that timing measurements can rarely yield more than a few bits of information per operation, since timing information is normally quantized to an integral number of clock cycles. In choosing L_{MAX}, the designer must assume that attackers will be able to combine information gleaned from multiple types of attacks. If the leakage rate is too large (as in the extreme case where L equals the key size because the entire key can be extracted during a single transaction), additional design features should be added to reduce L and reduce the value needed for L_{MAX}. Such additional measures can include known methods, such as filtering the device's power inputs, adding shielding, introducing noise into the timing or power consumption, implementing constant-time and constant execution path algorithms, and changing the device layout. Again, note that the designer of a leak-resistant system does not actually need to know what information is being revealed or how it is leaked; all he or she must do is choose an upper bound for the rate at which attackers might learn information about the keys. In contrast, the designer of a traditional system faces the much harder task of ensuring that no information about the secrets will leak out.

There are many ways information about secrets can leak from cryptosystems. For example, an attacker can use a high-speed analog-to-digital converter to record a smartcard's power consumption during a cryptographic operation. The amount of useful information that can be gained from such a measurement varies, but it would be fairly typical to gain enough information to guess each of 128 key bits correctly with a probability of 0.7. This information can reduce the amount of effort required for a brute force attack. For example, a brute force attack with one message against a key containing k bits where each bit's value is known with probability p can be completed in

$$E(k,p)=\sum_{i=0}^{k}\left[\binom{k}{i}(1-p)^{i}\,p^{k-i}\left[\left(\sum_{j=0}^{i}\binom{k}{j}\right)-\frac{1}{2}\binom{k}{i}\right]+\frac{1}{2}\right]$$

operations. The reduction in the effort for a brute force attack is equivalent to shortening the key by $L = \log_2(E(k,½) / E(k,p)) = \log_2(k - E(k,p) - 1)$ bits. (For example, in the case of $k = 128$ and $p = 0.7$, L is estimated to be about 11 bits for the first measurement. With a multiple message attack, the attacker's effort can fall to as low as $E(k,p)=\frac{1}{p^k}$.) Attackers can gain additional information about the keys by measuring additional operations; unless leak-proofing is used, finding the key becomes easy after just a few dozen operations.

When choosing L_{MAX}, a system designer should consider the signal-to-noise ratio of an attacker's measurements. For example, if the signal and noise are of roughly equivalent magnitude, the designer knows that an attacker's measurements should be incorrect about 25 percent of the time (e.g., $p = 0.75$ if only one observation per key bit is possible). Many measurement techniques, such as those involving timing, may have signal-to-noise ratios of 1:100 or worse. With such systems, L is generally quite small, but attackers who can make a large number of measurements can use averaging or other statistical techniques to recover the entire key. In extreme cases, attackers may be able to obtain all key bits with virtually perfect accuracy from a single transaction (i.e., $L = k$), necessitating the addition of shielding, noise in the power consumption (or elsewhere), and other measures to reduce p and L. Of course, L_{MAX} should be chosen conservatively; in the example above where less than 4 useful bits are obtained per operation for the given attack, the designer might select $L_{MAX} = 64$ for a leak-proof design.

When designing a traditional (i.e., non-leak-resistant) cryptosystem, a careful cryptosystem designer must study all possible information available to attackers if he or she is to ensure that no analytical techniques could be used to compromise the keys. In practice, many insecure systems are developed and deployed because such analysis is incomplete, too difficult even to attempt, or because the cryptographers working on the system do not understand or cannot completely control the physical characteristics of the device they are designing. Unexpected manufacturing defects or process changes, alterations made to the product by attackers, or modifications made to the product in the field can also introduce problems. Even a system designed and analyzed with great care can be broken if new or improved data collection

and analysis techniques are found later. In contrast, with leak-resistant cryptography, the system designer only needs to define an upper bound on the maximum rate at which attackers can extract information about the keys. A detailed understanding of the information available to attackers is not required, since leak-proof and leak-resistant cryptosystem designs allow for secret information in the device to leak out in (virtually) any way, yet remain secure despite this because leaked information is only of momentary value.

In a leak-proof design, with each new cryptographic operation i, the attacker is assumed to be able to choose any function F_i and determine the L_{MAX}-bit result of computing F_i on the device's secrets, inputs, intermediates, and outputs over the course of the operation. The attacker is even allowed to choose a new function F_i with each new operation. The system is considered leak-proof with a security factor n and leak rate L_{MAX} if, after observing a large number of operations, an attacker cannot forge signatures, decrypt data, or perform other sensitive operations without performing an exhaustive search to find an n-bit key or performing a comparable $O(2^n)$ operation. In addition to choosing L_{MAX}, designers also must choose n, and should select a value large enough to make exhaustive search infeasible.

Traditional Symmetric Authentication

Assume a user wishes to authenticate herself to a server using an n-bit secret key, K, known to both the server and the user's cryptographic token, but not known to attackers. The cryptographic token must be able to resist tampering to prevent, for example, attackers from being able to extract secrets from a stolen token. If the user's token has perfect tamper resistance (i.e., L=0), authentication protocols of the background art can be used. Typically the server sends a unique, unpredictable challenge value R to the user's token, which computes the value $A = H(R \| K)$, where "||" denotes concatenation and H is a one-way cryptographic hash function such as SHA. The user sends A to the server, which independently computes A (using its copy of K) and compares its result with the received value. The user authentication succeeds only if the comparison operation indicates a match.

If the function H is secure and if K is sufficiently large to prevent brute force attacks, attackers should not be able to obtain any useful information from the (R, A) values of old authentication sessions. To ensure that attackers cannot impersonate users by replaying old values

of A, the server generates values of R that are (with sufficiently high probability) unique. In most cases, the server should also make R unpredictable to ensure that an attacker with temporary possession of a token cannot compute future values of A. For example, R might be a 128-bit number produced using a secure random number generator (or pseudorandom number generator) in the server. The properties of cryptographic hash functions such as H have been the subject of considerable discussion in the literature. Hash functions typically provide functionality modeled after a random oracle, deterministically producing a particular output from any input. Ideally, such functions should be collision-resistant, non-invertable, should not leak partial information about the input from the output, and should not leak information about the output unless the entire input is known. Hash functions can have any output size. For example, MD5 produces 128-bit outputs and SHA produces 160-bit outputs. Hash functions may be constructed from other cryptographic primitives or other hash functions.

While the cryptographic security of the protocol using technology of the background art may be good, it is not leak-proof; even a one-bit leak function (with L=1) can reveal the key. For example, if the leak function F equals bit (R mod n) of K, an attacker can break the system quickly since a new key bit is revealed with every transaction where (R mod n) has a new value.

Leak-Proof Symmetric Authentication

The following leak-proof symmetric authentication protocol assumes a maximum leakage rate of L_{MAX} bits per transaction from the token. The user's token maintains a counter t, which is initialized to zero, and an (n+2L_{MAX})-bit shared secret K_t, which is initialized with a secret K_0. As in the traditional protocol, n is the security factor, meaning that attacks of complexity $O(2^n)$, such as brute-force of an n-bit key, are acceptable, but there should not be significantly easier attacks. (Note that against adversaries performing precomputation attacks based on Hellman's time/memory trade-off, larger values of n may be in order. Note also that some useful protocol security features, such as user and/or server identifiers in the hash operation inputs, have been omitted for simplicity in the protocol description. It is also assumed that no leaking will occur from the server. For simplicity in the protocol description, some possible security features (such as user and/or server identifiers in the hash operation inputs) have been omitted, and it is assumed that the server is in a physically secure environment.

As in the traditional protocol, the server begins the authentication process by generating a unique and unpredictable value R at step 105. For example, R might be a 128-bit output from a secure random number generator. At step 110, the server sends R to the user's token. At step 112, the token receives R. At step 115, the token increments t by computing $t \leftarrow t + 1$. At step 120, the token updates K_t by computing $K_t \leftarrow H_K(t \| K_t)$, where H_K is a cryptographic hash function that produces an $(n+2L_{MAX})$ bit output from the old value of K_t and the (newly incremented) value of t. Note that in the replacement operations (denoted "$\leftarrow$"), the token deletes the old values of t and K_t, replacing them with the new values. By deleting the old K_t, the token ensures that future leak functions cannot reveal information about the old (deleted) value. At step 122, the token uses the new values of t and K_t to compute $A = H_A(K_t \| t \| R)$. At step 125, the token sends both t and A to the server, which receives them at step 130. At step 135, the server verifies that t is acceptable (e.g., not too large but larger than the value received in the last successful authentication). If t is invalid, the server proceeds to step 175. Otherwise, at step 140, the server initializes i to zero and K_t' to K_0. At step 145, the server compares i with the received value of t, proceeding to step 160 if they are equal. Otherwise, at step 150, the server increments i by computing $i \leftarrow i + 1$. At step 155, the server computes $K_t' \leftarrow H_K(i \| K_t')$, then proceeds back to step 145. At step 160, the server computes $A' = H_A(K_t' \| t \| R)$. Finally, at step 165, the server compares A and A', where the authentication succeeds at step 170 if they match, or fails at 175 if they do not match.

If, at any time, any $2L_{MAX}$ (or fewer) bits of useful information about the secret are known to the attacker, there are still $(n+2L_{MAX})-2L_{MAX} = n$ or more unknown bits. These n bits of unknown information ensure that attacks will require $O(2^n)$ effort, corresponding to the desired security factor.

This leak-proof design assumes that at the beginning of any transaction the attacker may have L_{MAX} bits of useful information about the state of the token (e.g., K_t) that were obtained using the leak function F in previous operation. During the transaction, the attacker can gain an additional L_{MAX} bits of useful information from the token. However, the attacker should have no more than L_{MAX} bits of useful information about K_t at the end of the transaction. The property that attackers lose useful information during normal operation of the system is a characteristic of the leak-proof cryptosystem. In general, this information loss is achieved when the cryptosystem

performs operations that convert attackers' useful partial information about the secret into useless information. (Information is considered useless if it gives an attacker nothing better than the ability to test candidate values in an $O(2^n)$ exhaustive search or other "hard" operation. For example, if exhaustive search of X is hard and H is a good hash function, $H(X)$ is useless information to an attacker trying to find X.)

Thus, the attacker is assumed to begin with L_{MAX} bits of useful information about K_t before the token's $K_t \leftarrow H_K(t \,||\, K_t)$ computation. (Initial information about anything other than K_t is of no value to an attacker because K_t is the only secret value in the token. The algorithm H_K and the value of t are not assumed to be secret.) The attacker's information can be any function of K_t produced from the previous operation's leaks.

During the course of a transaction, the leak function F might reveal up to L_{MAX} information about the system and its secrets. Any information contained in the system may be leaked by F with only two restrictions: it should not reveal useful new information about values of K_t that were deleted before the operation started, and F should not reveal useful information about values of K_t that will be computed in future operations. These assumptions are completely reasonable, since real-world leaks would not reveal information about deleted or not-yet-existent data. The only way information about future K_t values could be leaked would be the bizarre case where the leak function itself included, or was somehow derived from, the function H_K. In practice, these assumptions about F are academic and of little concern. However, they are relevant when constructing proofs to demonstrate the security of a leak-proof system. For example, a hypothetical cryptosystem could theoretically leak information about $H_K(t + 3 \,||\, H_K(t + 2 \,||\, H_K(t + 1 \,||\, H_K(t \,||\, K_t))))$ during the first operation, about $H_K(t + 2 \,||\, H_K(t + 1 \,||\, H_K(t \,||\, K_t)))$ in the next, about $H_K(t + 1 \,||\, H_K(t \,||\, K_t))$ in the third, and about $H_K(t \,||\, K_t)$ in the fourth. Because t and K_t are updated during each operation, these expressions are all equal to $H_K(t \,||\, K_t)$ from the fourth operation, causing four leaks about a single value of K_t and providing $4L$ bits of useful information about K_t after the end of the fourth operation.

If the leak occurs at the beginning of the H_K computation, it could give the attacker up to $2L_{MAX}$ bits of useful information about the input value of K_t. Because K_t contains $(2L_{MAX}+n)$ bits of secret information and the attacker may have up to $2L_{MAX}$ bits of useful information about the initial value of K_t, there remain at least $(2L_{MAX}+n)-2L_{MAX} = n$ bits of information in K_t that are

secret. The hash function H_K effectively mixes up these n bits to produce a secure new K_t during each transaction such that the attacker's information about the old K_t is no longer useful.

If the leak occurs at the end of the H_K computation, it could give an attacker up to L_{MAX} bits of information about the final value of H_K, yielding L_{MAX} bits of information about the input to the subsequent transaction. This is not a problem, since the design assumes that attackers have up to L_{MAX} bits of information about K_t at the beginning of each transaction.

A third possibility is that the attacker's L_{MAX} bits of information might describe intermediates computed during the operation H_K. However, even if the attacker could obtain L_{MAX} new bits of information about the input to H_K and also L_{MAX} bits of information about the output from H_K, the system would be secure, since the attacker would never have more than $2L_{MAX}$ bits of information about the input K_t or more than L_{MAX} bits of information about the output K_t. Provided that L_{MAX} bits of information from within H_K cannot reveal more than L_{MAX} bits of information about the input, or more than L_{MAX} bits of information about the output, the system will be secure. This will be true unless H_K somehow compresses the input to form a short intermediate which is expanded to form the output. While hash functions whose internal states are smaller than their outputs should not be used, most cryptographic hash functions are fine.

A fourth possibility is that part or all of the leak could occur during the $A = H_A(K_t \| t \| R)$ calculation. The attacker's total "budget" for observations is L_{MAX} bits. If L_1 bits of leak occur during the H_K computation, an additional L_2 bits of information can leak during the $A = H_A(K_t \| t \| R)$ operation, where $L_2 \leq L_{MAX} - L_1$. If the second leak provides information about K_t, this is no different from leaking information about the result of the H_K computation; the attacker will still conclude the transaction with no more than L_{MAX} bits of information about K_t because $L_1 + L_2 \leq L_{MAX}$. However, the second leak could reveal information about A. If A must be kept secure against leaks (to prevent, for example, an attacker from using a leak to capture A and using A before the legitimate user can), the size of A should include an extra L_{MAX} bits (to provide security even if $L_2 = L_{MAX}$). Like H_K, H_A should not leak information about deleted or future values of K_t that are not used in or produced by the given operation. As with the similar assumptions on leaks from H_K, this limitation is primarily academic and of little practical concern, since real-world leak functions do not reveal information about deleted or not-yet-computed data. However, designers might be cautious when using unusual designs for H_A that are based on or derived from H_K,

particularly if the operation $H_A(K_t \| t \| R)$ could reveal useful information about the result of computing $H_K(t \| K_t)$.

Other Leak-Proof and Leak-Resistant Symmetric Schemes

Leak-proof symmetric data verification is often useful if, for example, a device needs to support symmetrically-signed code, data, or parameter updates. In existing systems, a hash or MAC of the data is typically computed using a secret key and the data is rejected if computed hash or MAC does not match a value received with the data. For example, a MAC may be computed as HMAC(K, data), where HMAC is defined in "RFC 2104, HMAC: Keyed-Hashing for Message Authentication" by H. Krawczyk, M. Bellare, and R. Canetti, 1997. Traditional (non-leak-resistant) designs are often vulnerable to attacks including power consumption analysis of MAC functions and timing analysis of comparison operations.

In a leak-proof verification protocol, the verifying device maintains a counter t and a key K_t, which are initialized (for example at the factory) with $t \leftarrow 0$ and $K_t \leftarrow K_0$. Before the transaction, the token provides t to the device providing the signed data ("signer"). The signer uses t to compute K_{t+1}' from K_0 using the relation $K_i = H_K(i \| K_{i-1})$, computes signature $S' =$ HMAC(K_{t+1}', data), and sends (data, S', t) to the verifying device. The verifier confirms that the received value of t matches its value of t, and rejects the signature if it does not. If t matches, the token increments t and updates K_t in its nonvolatile memory by computing $t \leftarrow t + 1$ and $K_t \leftarrow H_K(t \| K_t)$. (Note that in some applications, if the received value of t is larger than the internal value but the difference is not unreasonably large, it may be more appropriate to accept the signature and perform multiple updates to K_t instead of rejecting the signature outright.) Finally, the verifier computes $S =$ HMAC(K_t, data) and verifies that $S = S'$, rejecting the signature if S' does not equal the value of S received with the data.

Leak-resistant symmetric cryptography can be tailored to a wide variety of applications and environments. If data encryption is desired instead of authentication, the key K_t may be used for encryption. If communications between the device and the server are unreliable (for example if the server uses voice recognition or manual input to receive t and A), then small errors in the signature may be ignored. (One skilled in the art will appreciate that many functions may be used to determine whether a signature corresponds to its expected value.) The order of operations and

of data values may be adjusted, or additional steps and parameters may be added, without significantly changing the invention. To save on communication bandwidth or memory, the high order bits or digits of t may not need to be communicated or remembered. As a performance optimization, devices need not recompute K_t from K_0 with each new transaction. For example, when a transaction succeeds, the server can discard K_0 and maintain the validated version of K_t. If bi-directional authentication is required, the protocol can include a step whereby the server can authenticates itself to the user (or user's token) after the user's authentication is complete. If the server needs to be secured against leaks as well (as in the case where the role of "server" is played by an ordinary user), it can maintain its own counter t. In each transaction, the parties agree to use the larger of their two t values, where the device with the smaller t value performs extra updates to K_t to synchronize t. In an alternate embodiment for devices that contain a clock and a reliable power source (e.g., battery), the update operation may be performed periodically, for example by computing $K_t \leftarrow \mathrm{H_K}(t \,||\, K_t)$ once per second. The token uses the current K_t to compute $A = \mathrm{H_A}(K_t \,||\, t \,||\, R)$ or, if the token does not have any means for receiving R, it can output $A = \mathrm{H_A}(K_t)$. The server can use its clock and local copy of the secret to maintain its own version of K_t, which it can use to determine whether received values of A are recent and correct. The forgoing shows that the method and apparatus of the present invention can be implemented using numerous variations and modifications to the exemplary embodiments described herein, as would be known by one skilled in the art.

Traditional Certified Diffie-Hellman

While symmetric cryptosystems are sufficient for some applications, asymmetric cryptography is required for many applications. There are several ways leak resistance can be incorporated into public key cryptosystems, but in the ideal case it should have as little impact as possible on the overall system architecture. Most of the exemplary designs have thus been chosen to incorporate leak resistance into widely used cryptosystems in a way that only alters the key management device, and does not affect the certification process, certificate format, public key format, or algorithms for using the public key.

Typical protocols in the background art for performing certified Diffie-Hellman exponential key agreement involve two communicating users (or devices) and a certifying

authority (CA). The CA uses an asymmetric signature algorithm (such as DSA) to sign certificates that specify a user's public Diffie-Hellman parameters (the prime p and generator g), public key ($p^x \bmod g$, where x is the user's secret exponent), and auxiliary information (such as the user's identity, a description of privileges granted to the certificate holder, a serial number, expiration date, etc.). Certificates may be verified by anyone with the CA's public signature verification key. To obtain a certificate, user U typically generates a secret exponent (x_u), computes his or her own public key $y_u = g^{x_u} \bmod p$, presents y_u along with any required auxiliary identifying or authenticating information (e.g., a passport) to the CA, who issues the user a certificate C_u. Depending on the system, p and g may be unique for each user, or they may be system-wide constants (as will be assumed in the following description of Diffie-Hellman using the background art).

Using techniques of the background art, Alice and Bob can use their certificates to establish a secure communication channel. They first exchange certificates (C_{Alice} and C_{Bob}). Each verifies that the other's certificate is acceptable (e.g., properly formatted, properly signed by a trusted CA, not expired, not revoked, etc.). Because this protocol will assume that p and g are constants, they also check that the certificate's p and g match the expected values. Alice extracts Bob's public key (y_{Bob}) from C_{Bob} and uses her secret exponent (x_{Alice}) to compute $z_{\text{Alice}} = (y_{\text{Bob}})^{x_{\text{Alice}}} \bmod p$. Bob uses his secret exponent and Alice's public key to compute $z_{\text{Bob}} = (y_{\text{Alice}})^{x_{\text{Bob}}} \bmod p$. If everything works correctly, $z_{\text{Alice}} = z_{\text{Bob}}$, since:

$$\begin{aligned} z_{\text{Alice}} &= (y_{\text{Bob}})^{x_{\text{Alice}}} \bmod p \\ &= (g^{x_{\text{Bob}}})^{x_{\text{Alice}}} \bmod p \\ &= (g^{x_{\text{Alice}}})^{x_{\text{Bob}}} \bmod p \\ &= (y_{\text{Alice}})^{x_{\text{Bob}}} \bmod p \\ &= z_{\text{Bob}}. \end{aligned}$$

Thus, Alice and Bob have a shared key $z = z_{\text{Alice}} = z_{\text{Bob}}$. An attacker who pretends to be Alice but does not know her secret exponent (x_{Alice}) will not be able to compute $z_{\text{Alice}} = (y_{\text{Bob}})^{x_{\text{Alice}}} \bmod p$ correctly. Alice and Bob can positively identify themselves by showing that they correctly found z. For example, each can compute and send the other the hash of z

concatenated with their own certificate. Once Alice and Bob have verified each other, they can use a symmetric key derived from z to secure their communications. (For an example of a protocol in the background art that uses authenticated Diffie-Hellman, see "The SSL Protocol Version 3.0" by A. Freier, P. Karlton, and P. Kocher, March 1996.)

Leak-Resistant Certified Diffie-Hellman

A satisfactory leak-resistant public key cryptographic scheme must overcome the problem that, while certification requires the public key be constant, information about the corresponding private key should not leak out of the token that contains it. In the leak-proof symmetric protocol described above, the design assumes that the leak function reveals no useful information about old deleted values of K_t or about future values of K_t that have not yet been computed. Existing public key schemes, however, require that implementations repeatedly perform a consistent, usually deterministic, operation using the private key. For example, in the case of Diffie-Hellman, a leak-resistant token that is compatible with existing protocols and implementations must be able to perform the secret key operation $y^x \bmod p$, while ensuring that the exponent x remains secret. The radical reshuffling of the secret provided by the hash function H_K in the symmetric approach cannot be used because the device must be able to perform the same operation consistently.

The operations used by the token to perform the private key operation are modified to add leak resistance using the following variables:

Register	Comment
x_1	First part of the secret key (in nonvolatile updateable memory)
x_2	Second part of the secret key (in nonvolatile updateable memory)
g	The generator (not secret).
p	The public prime, preferably a strong prime (not secret).

The prime p and generator g may be global parameters, or may be specific to individual users or groups of users (or tokens). In either case, the certificate recipient must be able to obtain p and g securely, usually as built-in constants or by extracting them from the certificate.

To generate a new secret key, the key generation device (often but not always the cryptographic token that will contain the key) must first obtain or generate p and g, where p is the prime and g is a generator mod p. If p and g are not system-wide parameters, algorithms known

in the background art for selecting large prime numbers and generators may be used. It is recommended that p be chosen with $\frac{p-1}{2}$ also prime, or at least that $\phi(p)$ not be smooth. (When $\frac{p-1}{2}$ is not prime, information about x_1 and x_2 modulo small factors of $\phi(p)$ may be leaked, which is why it is preferable that $\phi(p)$ not be smooth. Note that ϕ denotes Euler's totient function.) Once p and g have been chosen, the device generates two random exponents x_1 and x_2. The lowest-order bit of x_1 and of x_2 is not considered secret, and may be set to 1. Using p, g, x_1, and x_2, the device can then compute its public key as $g^{x_1 x_2} \bmod p$ and submit it, along with any required identifying information or parameters needed (e.g., p and g), to the CA for certification.

Figure 2 illustrates the process followed by the token to perform private key operations. At step 205, the token obtains the input message y, its own (non-secret) prime p, and its own secret key halves (x_1 and x_2). If x_1, x_2, and p are stored in encrypted and/or authenticated form, they would be decrypted or verified at this point. At this step, the token should verify that $1 < y < p-1$. At step 210, the token uses a random number generator (or pseudorandom number generator) to select a random integer b_0, where $0 < b_0 < p$. At step 215, the token computes $b_1 = b_0^{-1} \bmod p$. The inverse computation mod p may be performed using the extended Euclidian algorithm or the formula $b_1 = b_0^{\phi(p)-1} \bmod p$. At step 220, the token computes $b_2 = b_1^{x_1} \bmod p$. At this point, b_1 is no longer needed; its storage space may be used to store b_2. Efficient algorithms for computing modular exponentiation, widely known in the art, may be used to complete step 220. Alternatively, when a fast modular exponentiator is available, the computation b_2 may be performed using the relationship $b_2 = b_0^{\phi(p)-x_1} \bmod p$. At step 225, the token computes $b_3 = b_2^{x_2} \bmod p$. At this point, b_2 is no longer needed; its storage space may be used to store b_3. At step 230, the token computes $z_0 = b_0 y \bmod p$. At this point, y and b_0 are no longer needed; their space may be used to store r_1 (computed at step 235) and z_0. At step 235, the token uses a random number generator to select a random integer r_1, where $0 < r_1 < \phi(p)$ and $\gcd(r_1, \phi(p)) = 1$. (If $\frac{p-1}{2}$ is known to be prime, it is sufficient to verify that r_1 is odd.) At step 240, the token updates x_1 by computing $x_1 \leftarrow x_1 r_1 \bmod \phi(p)$. The old value of x_1 is deleted and replaced with the updated value. At step 245, the token computes $r_2 = (r_1^{-1}) \bmod \phi(p)$. If $\frac{p-1}{2}$ is prime, then r_2 can be found using a modular exponentiator and the Chinese Remainder Theorem. Note that r_1 is not

needed after this step, so its space may be used to store r_2. At step 250, the token updates x_2 by computing $x_2 \leftarrow x_2 r_2 \bmod \phi(p)$. The old value of x_2 should be deleted and replaced with the updated value. At step 255, the token computes $z_1 = (z_0)^{x_1} \bmod p$. Note that z_0 is not needed after this step, so its space may be used to store z_1. At step 260, the token computes $z_2 = (z_1)^{x_2} \bmod p$. Note that z_1 is not needed after this step, so its space may be used to store z_2. At step 265, the token finds the exponential key exchange result by computing $z = z_2 b_3 \bmod p$. Finally, at step 270, the token erases and frees any remaining temporary variables.

The process shown in Figure 2 correctly computes $z = y^x \bmod p$, where $x = x_1 x_2 \bmod \phi(p)$, since:

$$\begin{aligned} z &= z_2 b_3 \bmod p \\ &= \left(z_1^{x_2} \bmod p\right)\left(b_2^{x_2} \bmod p\right) \bmod p \\ &= \left(\left(z_0^{x_1} \bmod p\right)^{x_2}\right)\left(\left(b_1^{x_1} \bmod p\right)^{x_2}\right) \bmod p \\ &= \left(b_0 y \bmod p\right)^{x_1 x_2}\left(b_0^{-1} \bmod p\right)^{x_1 x_2} \bmod p \\ &= y^{x_1 x_2} \bmod p \\ &= y^x \bmod p. \end{aligned}$$

The invention is useful for private key owners communicating with other users (or devices) who have certificates, and also when communicating with users who do not. For users or devices lacking certificates or public keys that must be constant, secure long-term secret storage is usually not required since uncertified Diffie-Hellman may be used.

If Alice has a certificate and wishes to communicate with Bob who does not have a certificate, the protocol proceeds as follows. Alice sends her certificate (C_{Alice}) to Bob, who receives it and verifies that it is acceptable. Bob extracts y_{Alice} (along with p_{Alice} and g_{Alice}, unless they are system-wide parameters) from C_{Alice}. Next, Bob generates a random exponent x_{BA}, where $0 < x_{\text{AB}} < \phi(p_{\text{Alice}})$. Bob then uses his exponent x_{AB} and Alice's parameters to calculate $y_{\text{BA}} = \left(g_{\text{Alice}}^{x_{\text{BA}}}\right) \bmod p_{\text{Alice}}$ and the session key $z = \left(y_{\text{Alice}}^{x_{\text{BA}}}\right) \bmod p_{\text{Alice}}$. Bob sends y_{BA} to Alice, who performs the operation illustrated in Figure 2 to update her internal parameters and derive z from y_{BA}. Alice then proves that she computed z correctly, for example by sending Bob H(z || C_{Alice}). (Alice cannot authenticate Bob because he does not have a certificate. Consequently, she

does not necessarily need to verify that he computed z successfully.) Finally, Alice and Bob can use z (or, more commonly, a key derived from z) to secure their communications.

If both Alice and Bob have certificates, the protocol works as follows. First, Alice and Bob exchange certificates (C_{Alice} and C_{Bob}), and each verifies that other's certificate is valid. Alice then extracts the parameters p_{Bob}, g_{Bob}, and y_{Bob} from C_{Bob}, and Bob extracts p_{Alice}, g_{Alice}, and y_{Alice} from C_{Alice}. Alice then generates a random exponent x_{AB} where $0 < x_{AB} < \phi(p_{Bob})$, computes $y_{AB} = (g_{Bob})^{x_{AB}} \bmod p_{Bob}$, and computes $z_{AB} - (y_{Bob})^{x_{AB}} \bmod p_{Bob}$. Bob generates a random x_{BA} where $0 < x_{BA} < \phi(p_{Alice})$, computes $y_{BA} = (g_{Alice})^{x_{BA}} \bmod p_{Alice}$, and computes $z_{BA} - (y_{Alice})^{x_{BA}} \bmod p_{Alice}$. Bob sends y_{BA} to Alice, and Alice sends y_{AB} to Bob. Alice and Bob each perform the operation shown in Figure 2, where each uses the prime p from their own certificate and their own secret exponent halves (x_1 and x_2). For the message y in Figure 2, Alice uses y_{BA} (received from Bob), and Bob uses y_{AB} (received from Alice). Using the process shown in Figure 2, Alice computes z. Using z and z_{AB} (computed previously), she can find a session key K. This may be done, for example, by using a hash function H to compute $K = H(z \,||\, z_{AB})$. The value of z Bob obtains using the process shown in Figure 2 should equal Alice's z_{AB}, and Bob's z_{BA} (computed previously) should equal Alice's z. If there were no errors or attacks, Bob should thus be able to find K, e.g., by computing $K = H(z_{BA} \,||\, z)$. Alice and Bob now share K. Alice can prove her identity by showing that she computed K correctly, for example by sending Bob $H(K \,||\, C_{Alice})$. Bob can prove his identity by sending Alice $H(K \,||\, C_{Bob})$. Alice and Bob can then secure their communications by encrypting and authenticating using K or a key derived from K.

Note that this protocol, like the others, is provided as an example only; many variations and enhancements of the present invention are possible and will be evident to one skilled in the art. For example, certificates may come from a directory, more than two parties can participate in the key agreement, key escrow functionality may be added, the prime modulus p may be replaced with a composite number, etc. Note also that Alice and Bob as they are called in the protocol are not necessarily people; they would normally be computers, cryptographic devices, etc.

For leak resistance to be effective, attackers must not be able to gain new useful information about the secret variables with each additional operation unless a comparable amount of old useful information is made useless. While the symmetric design is based on the assumption

that leaked information will not survive the hash operation H_K, this design uses multiplication operations mod $\phi(p)$ to update x_1 and x_2. The most common variety of leaked information, statistical information about exponent bits, is not of use to attackers in this design, as the exponent update process ($x_1 \leftarrow x_1 r_1 \bmod \phi(p)$ and $x_2 \leftarrow x_2 r_2 \bmod \phi(p)$) destroys the utility of this information. The only relevant characteristic that survives the update process is that $x_1 x_2 \bmod \phi(p)$ remains constant, so the system designer must be careful to ensure that the leak function does not reveal information allowing the attacker to find new useful information about $x_1 x_2 \bmod \phi(p)$.

There is a modest performance penalty, approximately a factor of four, for the leak-resistant design as described. One way to improve performance is to remove the blinding and unblinding operations, which are often unnecessary. (The blinding operations prevent attackers from correlating input values of y with the numbers processed by the modular exponentiation operation.) Alternatively or additionally, it is possible to update and reuse values of b_0, b_3, r_1, and r_2 by computing $b_0 \leftarrow (b_0)^v \bmod p$, $b_3 \leftarrow (b_3)^v \bmod p$, $r_1 \leftarrow (r_1)^w \bmod \phi(p)$, and $r_2 \leftarrow (r_2)^w \bmod \phi(p)$, where v and w are fairly short random exponents. Note that the relationship $b_3 \leftarrow b_0^{-x_1 x_2} \bmod p$ remains true when b_0 and b_3 are both raised to the power $v \pmod p$. The relationship $r_2 = (r_1^{-1}) \bmod \phi(p)$ also remains true when r_1 and r_2 are exponentiated (mod $\phi(p)$). Other parameter update operations may also be used, such as exponentiation with fixed exponents (e.g., $v = w = 3$), or multiplication with random values and their inverses, mod p and $\phi(p)$. The time per transaction with this update process is about half that of the unoptimized leak-resistant implementation, but additional storage is required and care must be taken to ensure that b_0, b_3, r_1, and r_2 will not be leaked or otherwise compromised.

It should also be noted that with this particular type of certified Diffie-Hellman, the negotiated key is the same every time any given pair of users communicate. Consequently, though the blinding operation performed using b_0 and b_3 does serve to protect the exponents, no algorithm can ensure that the result K will not be leaked in the final step or by the system after the process is complete. If storage is available, parties could keep track of the values of y they have received (or their hashes) and reject duplicates. Alternatively, to ensure that a different result is obtained from each negotiation, Alice and Bob can generate and exchange additional exponents, w_{Alice} and w_{Bob}, for example with $0 < w < 2^{128}$ (where $2^{128} << p$). Alice sets $y = (y_{BA})^{w_{Alice} w_{Bob}} \bmod p$

instead of just $y = y_{BA}$, and Bob sets $y = (y_{AB})^{w_{Bob} w_{Alice}} \bmod p$ instead of $y = y_{AB}$ before performing the operation shown in Figure 2.

Leak-Resistant RSA

To give RSA private key operations resistance to leaks, it is possible to divide the exponent into two halves such that information about either half is destroyed with each operation. Private key signing operations with the RSA algorithm involve computing $S = M^d \bmod n$, where M is the message, S is the signature (verifiable using $M = S^e \bmod n$), $d = e^{-1} \bmod \phi(n)$, and $n = pq$, where n and e are public and p and q are secret primes. For RSA to be secure, d, $\phi(n)$, p, and q must all be secret. Private key RSA decryption operations are virtually identical to signing, since the decrypted message M is recovered from the ciphertext C by computing $M = C^d \bmod n$. Although the following discussion uses variable names from the RSA signing operation, the same techniques may be applied identically to decryption.

A leak-resistant scheme for RSA implementations may be constructed as illustrated in Figure 3. At step 300, prior to the commencement of any signing or decryption operations, the device is initialized with (or creates) the public and private keys. The device contains the public modulus n and the secret key components d_1, d_2, and z, and k, where k is a prime number of medium-size (e.g., $0 < k < 2^{128}$) chosen at random, $z = k\phi(n)$, d_1 is a random number such that $0 < d_1 < z$ and $\gcd(d_1, z) = 1$, and $d_2 = (e^{-1} \bmod \phi(n))(d_1^{-1} \bmod z) \bmod z$. Techniques for generating the initial RSA primes (e.g., p and q) and modulus (n) are well known in the background art. At step 305, the device computes a random prime k' of medium size (e.g., $0 < k' < 2^{128}$). (Algorithms for efficiently generating prime numbers are known in the art.)

At step 303, the device (token) receives a message M to sign (or to decrypt). At step 310, the device updates z by computing $z \leftarrow k'z$. At step 315, the device updates z again by computing $z \leftarrow z / k$. (There should be no remainder from this operation, since k divides z.) At step 320, k is replaced with k' by performing $k \leftarrow k'$. Because k' will not be used in subsequent operations, its storage space may be used to hold R (produced at step 325). At step 325, the device selects a random R where $0 < R < z$ and $\gcd(R, z) = 1$. At step 330, the device updates d_1 by computing $d_1 \leftarrow d_1 R \bmod z$. At step 335, the device finds the inverse of R by computing $R' \leftarrow R^{-1} \bmod z$ using, for example, the extended Euclidian algorithm. Note that R is no longer needed after this step, so

its storage space may be erased and used to hold R'. At step 340, the device updates d_2 by computing $d_2 \leftarrow d_2 R' \bmod z$. At step 345, the device computes $S_0 = M^{d_1} \bmod n$, where M is the input message to be signed (or the message to be decrypted). Note that M is no longer needed after this step, so its storage space may be used for S_0. At step 350, the device computes $S = S_0^{d_2} \bmod n$, yielding the final signature (or plaintext if decrypting a message). Leak-resistant RSA has similar security characteristics as normal RSA; standard message padding, post-processing, and key sizes may be used. Public key operations are also performed normally (e.g., $M = S^e \bmod n$).

A simpler RSA leak resistance scheme may be implemented by splitting the exponent d into two halves d_1 and d_2 such that $d_1 + d_2 = d$. This can be achieved during key generation by choosing d_1 to be a random integer where $0 \le d_1 \le d$, and choosing $d_2 \leftarrow d - d_1$. To perform private key operations, the device needs d_1 and d_2, but it does not need to contain d. Prior to each private key operation, the cryptographic device identifies which of d_1 and d_2 is larger. If $d_1 > d_2$, then the device computes a random integer r where $0 \le r \le d_1$, adds r to d_2 (i.e., $d_2 \leftarrow d_2 + r$), and subtracts r from d_1 (i.e., $d_1 \leftarrow d_1 - r$). Otherwise, if $d_1 \le d_2$, then the device chooses a random integer r where $0 \le r \le d_2$, adds r to d_1 (i.e., $d_1 \leftarrow d_1 + r$), and subtracts r from d_2 (i.e., $d_2 \leftarrow d_2 - r$). Then, to perform the private key operation on a message M, the device computes $s_1 = M^{d_1} \bmod n$, $s_2 = M^{d_2} \bmod n$, and computes the signature $S = s_1 s_2 \bmod n$. While this approach of splitting the exponent into two halves whose sum equals the exponent can also be used with Diffie-Hellman and other cryptosystems, dividing the exponent into the product of two numbers mod ϕ(p) is usually preferable since the assumption that information about $d_1 + d_2$ will not leak is less conservative than the assumption that information about $x_1 x_2 \bmod \phi(p)$ will not leak. In the case of RSA, updates mod $\phi(n)$ cannot be done safely, since $\phi(n)$ must be kept secret.

When the Chinese Remainder Theorem is required for performance, it is possible to use similar techniques to add leak resistance by maintaining multiples of the secret primes (p and q) that are updated every time (e.g., multiplying by the new multiple then dividing by the old multiple). These techniques also protect the exponents (d_p and d_q) as multiples of their normal values. At the end of the operation, the result S is corrected to compensate for the adjustments to d_p, d_q, p, and q. In some cases, other techniques may also be appropriate. For example, exponent

vector codings may be rechosen frequently using, for example, a random number generator. Also, Montgomery arithmetic may be performed mod j where j is a value that is changed with each operation (as opposed to traditional Montgomery implementations where j is constant with $j = 2^k$). The forgoing shows that the method and apparatus of the present invention can be implemented using numerous variations and modifications to the exemplary embodiments described herein, as would be known by one skilled in the art.

Leak-Resistant ElGamal Public Key Encryption and Digital Signatures

The private key in the ElGamal public key encryption scheme is a randomly selected secret a where $1 \le a \le p-2$. The non-secret parameters are a prime p, a generator α, and $\alpha^a \bmod p$. To encrypt a message m, one selects a random k (where $1 \le k \le p-2$) and computes the ciphertext (γ, δ) where $\gamma = \alpha^k \bmod p$ and $\delta = m(\alpha^a \bmod p)^k \bmod p$. Decryption is performed by computing $m = \delta(\gamma^{p-1-a}) \bmod p$. (See the Handbook of Applied Cryptography by A. Menezes, P. van Oorschot, and S. Vanstone, 1997, pages 294-298, for a description of ElGamal public-key encryption).

To make the ElGamal public-key decryption algorithm leak-resistant, the secret exponent $(p-1-a)$ is stored in two halves a_1 and a_2, such that $a_1 a_2 = (\phi(p)-a) \bmod \phi(p)$. When generating ElGamal parameters for this leak-resistant implementation, it is recommended, but not required, that p be chosen with $\frac{p-1}{2}$ prime so that $\phi(p)/2$ is prime. The variables a_1 and a_2 are normally chosen initially as random integers between 0 and $\phi(p)$. Alternatively, it is possible to generate a first, then choose a_1 and a_2, as by selecting a_1 relatively prime to $\phi(p)$ and computing $a_2 = (a^{-1} \bmod \phi(p))(a_1^{-1} \bmod \phi(p)) \bmod \phi(p)$.

Figure 4 illustrates the leak-resistant ElGamal decryption process. At step 405, the decryption device receives an encrypted message pair (γ, δ). At step 410, the device selects a random r_1 where $1 \le r_1 < \phi(p)$ and $\gcd(r_1, \phi(p)) = 1$. At step 415, the device updates a_1 by computing $a_1 \leftarrow a_1 r_1 \bmod \phi(p)$, over-writing the old value of a_1 with the new value. At step 420, the device computes the inverse of r_1 by computing $r_2 = (r_1)^{-1} \bmod \phi(p)$. Because r_1 is not used after this step, its storage space may be used to hold r_2. Note that if $\frac{p-1}{2}$ is prime, then r_2 may also be found by finding $r_2' = r_1^{(p-1)/2-2} \bmod \frac{p-1}{2}$, and using the CRT to find $r_2 \pmod{p-1}$. At step 425, the device updates a_2 by computing $a_2 \leftarrow a_2 r_2 \bmod \phi(p)$. At step 430, the device begins the

private key (decryption) process by computing $m' = \gamma^{a_1} \bmod p$. At step 435, the device computes $m = \delta(m')^{a_2} \bmod p$ and returns the message m. If verification is successful, the result equals the original message because:

$$\begin{aligned}(\delta)(m')^{a_2} \bmod p &= \left(m(\alpha^a)^k\right)\left(\gamma^{a_1} \bmod p\right)^{a_2} \bmod p \\ &= \left(m\alpha^{ak}\right)\left(\gamma^{a_1 a_2 \bmod \phi(p)}\right) \bmod p \\ &= \left(m\alpha^{ak}\right)\left((\alpha^k \bmod p)^{-a \bmod \phi(p)}\right) \bmod p \\ &= \left(m\alpha^{ak}\right)\left(\alpha^{-ak}\right) \bmod p \\ &= m\end{aligned}$$

As with the ElGamal public key encryption scheme, the private key for the ElGamal digital signature scheme is a randomly-selected secret a, where $1 \le a \le p-2$. The public key is also similar, consisting of a prime p, a generator α, and public parameter y where $y = \alpha^a \bmod p$. To sign a message m, the private key holder chooses or precomputes a random secret integer k (where $1 \le k \le p-2$ and k is relatively prime to $p-1$) and its inverse, $k^{-1} \bmod \phi(p)$. Next, the signer computes the signature (r, s), where $r = \alpha^k \bmod p$, $s = \left(\left(k^{-1} \bmod \phi(p)\right)\left[\mathrm{H}(m) - ar\right]\right) \bmod \phi(p)$, and $\mathrm{H}(m)$ is the hash of the message. Signature verification is performed using the public key (p, α, y) by verifying that $1 \le r < p$ and by verifying that $y^r r^s \bmod p = \alpha^{\mathrm{H}(m)} \bmod p$.

To make the ElGamal digital signing algorithm leak-resistant, the token containing the private key maintains three persistent variables, a_k, w, and r. Initially, $a_k = a$ (the private exponent), $w = 1$, and $r = \alpha$. When a message m is to be signed (or during the precomputation before signing), the token generates a random number b and its inverse $b^{-1} \bmod \phi(p)$, where b is relatively prime to $\phi(p)$ and $0 < b < \phi(p)$. The token then updates a_k, w, and r by computing $a_k \leftarrow (a_k)(b^{-1}) \bmod \phi(p)$, $w \leftarrow (w)(b^{-1}) \bmod \phi(p)$, and $r \leftarrow (r^b) \bmod p$. The signature (r, s) is formed from the updated value of r and s, where $s = \left(w(\mathrm{H}(m) - a_k r)\right) \bmod \phi(p)$. Note that a_k, w, and r are not randomized prior to the first operation, but should be randomized before exposure to possible attack, since otherwise the first operation may leak more information than subsequent ones. It is thus recommended that a dummy signature or parameter update with $a_k \leftarrow (a_k)(b^{-1}) \bmod \phi(p)$, $w \leftarrow (w)(b^{-1}) \bmod \phi(p)$, and $r \leftarrow (r^b) \bmod p$ be performed immediately after key generation. Valid signatures produced using the tamper-resistant ElGamal algorithm may be checked using the normal ElGamal signature verification algorithm.

It is also possible to split all or some the ElGamal variables into two halves as part of the leak resistance scheme. In such a variant, a is replaced with a_1 and a_2, w with w_1 and w_2, and r with r_1 and r_2. It is also possible to reorder the operations by performing, for example, the parameter updates as a precomputation step prior to receipt of the enciphered message. Other variations and modifications to the exemplary embodiments described herein will be evident to one skilled in the art.

Leak-Resistant DSA

The Digital Signature Algorithm (DSA, also known as the Digital Signature Standard, or DSS) is defined in "Digital Signature Standard (DSS)," Federal Information Processing Standards Publication 186, National Institute of Standards and Technology, May 19, 1994 and described in detail in the Handbook of Applied Cryptography, pages 452 to 454. In non-leak-proof systems, the private key consists of a secret parameter a, and the public key consists of (p, q, α, y), where p is a large (usually 512 to 1024 bit) prime, q is a 160-bit prime, α is a generator of the cyclic group of order q mod p, and $y = \alpha^a \bmod p$. To sign a message whose hash is $H(m)$, the signer first generates (or precomputes) a random integer k and its inverse k^{-1} mod q, where $0 < k < q$. The signer then computes the signature (r, s), where $r = (\alpha^k \bmod p) \bmod q$, and $s = (k^{-1} \bmod q)(H(m) + ar) \bmod q$.

To make the DSA signing algorithm leak-resistant, the token containing the private key maintains two variables in nonvolatile memory, a_k and k, which are initialized with $a_k = a$ and $k = 1$. When a message m is to be signed (or during the precomputation before signing), the token generates a random integer b and its inverse b^{-1} mod q, where $0 < b < q$. The token then updates a_k and k by computing $a_k \leftarrow (a_k b^{-1} \bmod q)(k) \bmod q$, followed by $k \leftarrow b$. The signature (r, s) is formed from the updated values of a_k and k by computing $r = \alpha^k \bmod p$, and $s = [(b^{-1}H(m) \bmod q) + (a_k r) \bmod q] \bmod q$. As indicated, when computing s, $b^{-1}H(m) \bmod q$ and $(a_k r) \bmod q$ are computed first, then combined mod q. Note that a_k and k should be randomized prior to the first operation, since the first update may leak more information than subsequent updates. It is thus recommended that a dummy signature (or parameter update) be performed immediately after key generation. Valid signatures produced using the leak-resistant DSA algorithm may be checked using the normal DSA signature verification procedure.

Other Algorithms and Applications

Other algorithms can be made leak-proof or leak-resistant, or may be incorporated into leak-resistant cryptosystems. For example, cryptosystems such as those based on elliptic curves (including elliptic curve analogs of other cryptosystems), secret sharing schemes, anonymous electronic cash protocols, threshold signatures schemes, etc. be made leak resistant using the present invention.

Implementation details of the schemes described may be adjusted without materially changing the invention, for example by re-ordering operations, inserting steps, substituting equivalent or similar operations, etc. Also, while new keys are normally generated when a new system is produced, it is often possible to add leak resistance retroactively while maintaining or converting existing private keys.

Leak-resistant designs avoid performing repeated mathematical operations using non-changing (static) secret values, since they are likely to leak out. However, in environments where it is possible to implement a simple function (such as an exclusive OR) that does not leak information, it is possible use this function to implement more complex cryptographic operations.

While the exemplary implementations assume that the leak functions can reveal any information present in the system, designers may often safely use the (weaker) assumption that information not used in a given operation will not be leaked by that operation. Schemes using this weaker assumption may contain a large table of precomputed subkey values, from which a unique or random subset are selected and/or updated for each operation. For example, DES implementations may use indexed permutation lookup tables in which a few table elements are exchanged with each operation.

While leak resistance provides many advantages, the use of leak resistance by itself cannot guarantee good security. For example, leak-resistant cryptosystems are not inherently secure against error attacks, so operations should be verified. (Changes can even be made to the cryptosystem and/or leak resistance operations to detect errors.) Similarly, leak resistance by itself does not prevent attacks that extract the entire state out of a device (e.g., $L=L_{MAX}$). For example, traditional tamper resistance techniques may be required to prevent attackers from staining ROM

or EEPROM memory cells and reading the contents under a microscope. Implementers should also be aware of interruption attacks, such as those that involve disconnecting the power or resetting a device during an operation, to ensure that secrets will not be compromised or that a single leaky operation will not be performed repeatedly. (As a countermeasure, devices can increment a counter in nonvolatile memory prior to each operation, and reset or reduce the counter value when the operation completes successfully. If the number of interrupted operations since the last successful update exceeds a threshold value, the device can disable itself.) Other tamper resistance mechanisms and techniques, such as the use of fixed-time and fixed-execution path algorithms for critical operations, may need to be used in conjunction with leak resistance, particularly for systems with a relatively low self-healing rate (e.g., L_{MAX} is small).

Leak-resistant algorithms, protocols, and devices may be used in virtually any application requiring cryptographic security and secure key management, including without limitation: smartcards, electronic cash, electronic payments, funds transfer, remote access, timestamping, certification, certificate validation, secure e-mail, secure facsimile, telecommunications security (voice and data), computer networks, radio and satellite communications, infrared communications, access control, door locks, wireless keys, biometric devices, automobile ignition locks, copy protection devices, payment systems, systems for controlling the use and payment of copyrighted information, and point of sale terminals.

The forgoing shows that the method and apparatus of the present invention can be implemented using numerous variations and modifications to the exemplary embodiments described herein, as would be known by one skilled in the art. Thus, it is intended that the scope of the present invention be limited only with regard to the claims below.

FIG. 1

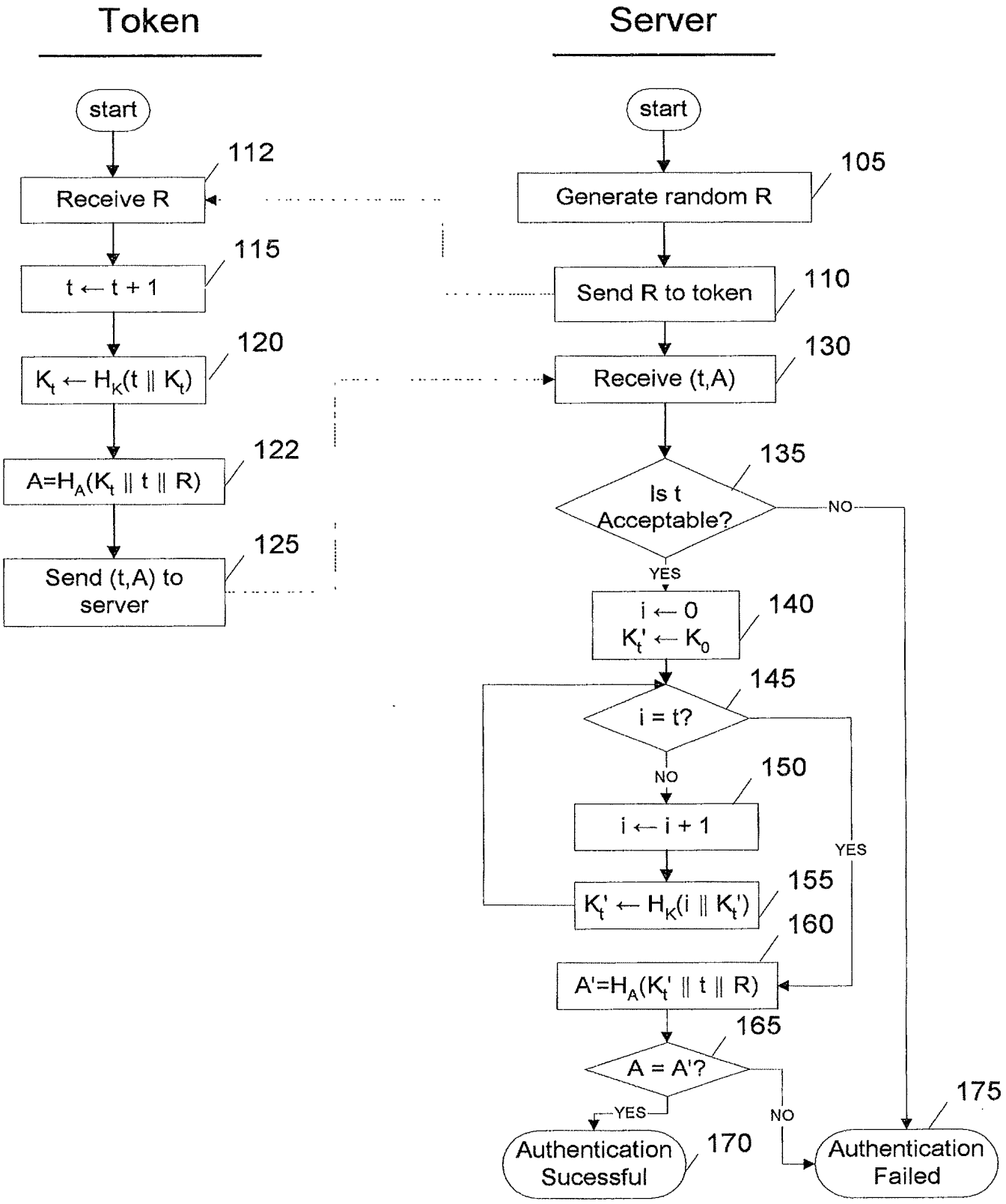

Token
Server
start
Receive R
112
t ← t + 1
115
$K_t \leftarrow H_K(t \| K_t)$
120
$A=H_A(K_t \| t \| R)$
122
Send (t,A) to server
125
start
Generate random R
105
Send R to token
110
Receive (t,A)
130
Is t Acceptable?
135
NO
YES
$i \leftarrow 0$
$K_t' \leftarrow K_0$
140
i = t?
145
NO
YES
$i \leftarrow i + 1$
150
$K_t' \leftarrow H_K(i \| K_t')$
155
$A'=H_A(K_t' \| t \| R)$
160
A = A'?
165
YES
NO
Authentication Sucessful
170
Authentication Failed
175

FIG. 2

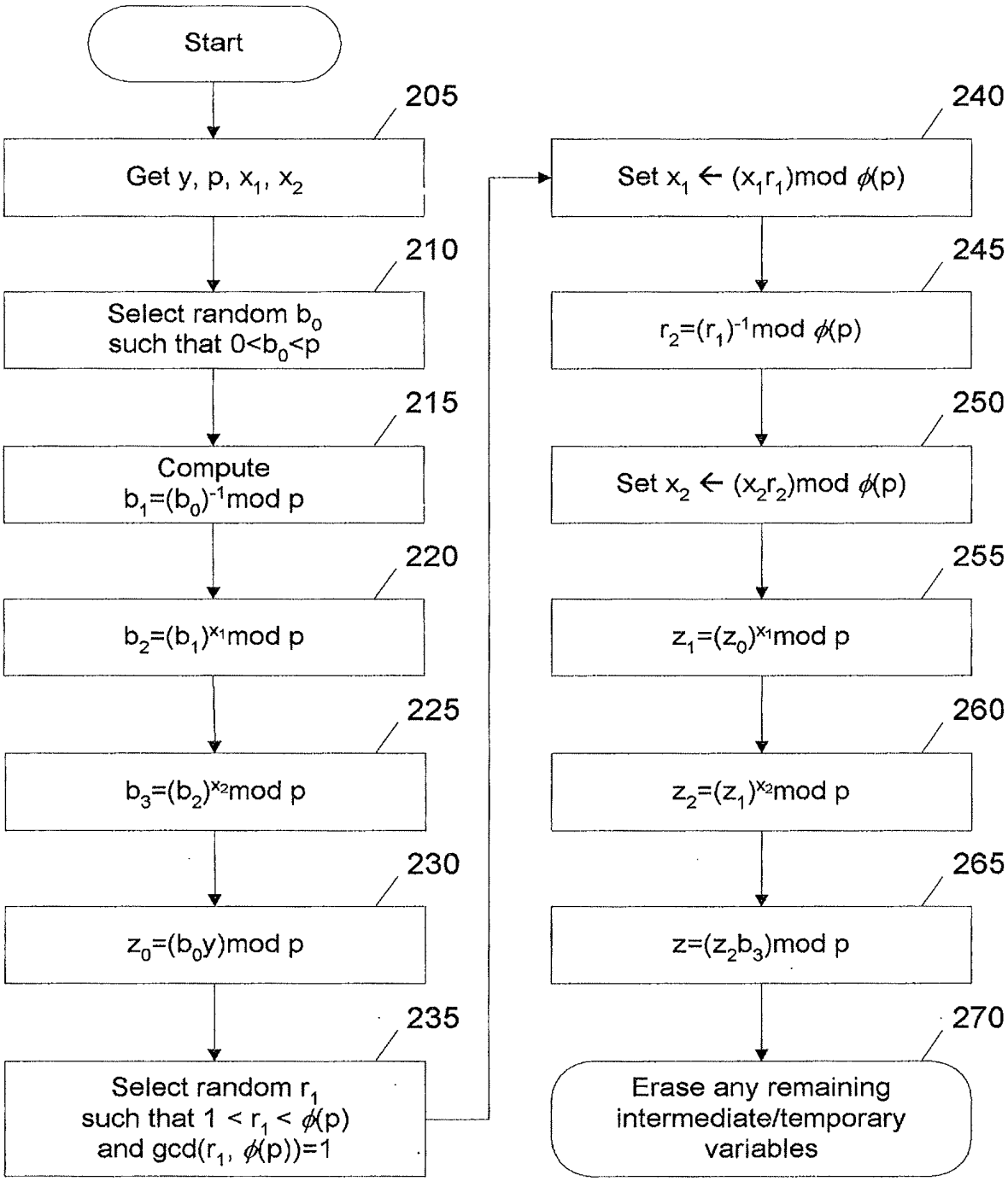

Start
205
Get y, p, x_1, x_2
210
Select random b_0 such that $0<b_0<p$
215
Compute $b_1=(b_0)^{-1} \bmod p$
220
$b_2=(b_1)^{x_1} \bmod p$
225
$b_3=(b_2)^{x_2} \bmod p$
230
$z_0=(b_0 y) \bmod p$
235
Select random r_1 such that $1 < r_1 < \phi(p)$ and $\gcd(r_1, \phi(p))=1$
240
Set $x_1 \leftarrow (x_1 r_1) \bmod \phi(p)$
245
$r_2=(r_1)^{-1} \bmod \phi(p)$
250
Set $x_2 \leftarrow (x_2 r_2) \bmod \phi(p)$
255
$z_1=(z_0)^{x_1} \bmod p$
260
$z_2=(z_1)^{x_2} \bmod p$
265
$z=(z_2 b_3) \bmod p$
270
Erase any remaining intermediate/temporary variables

FIG. 3

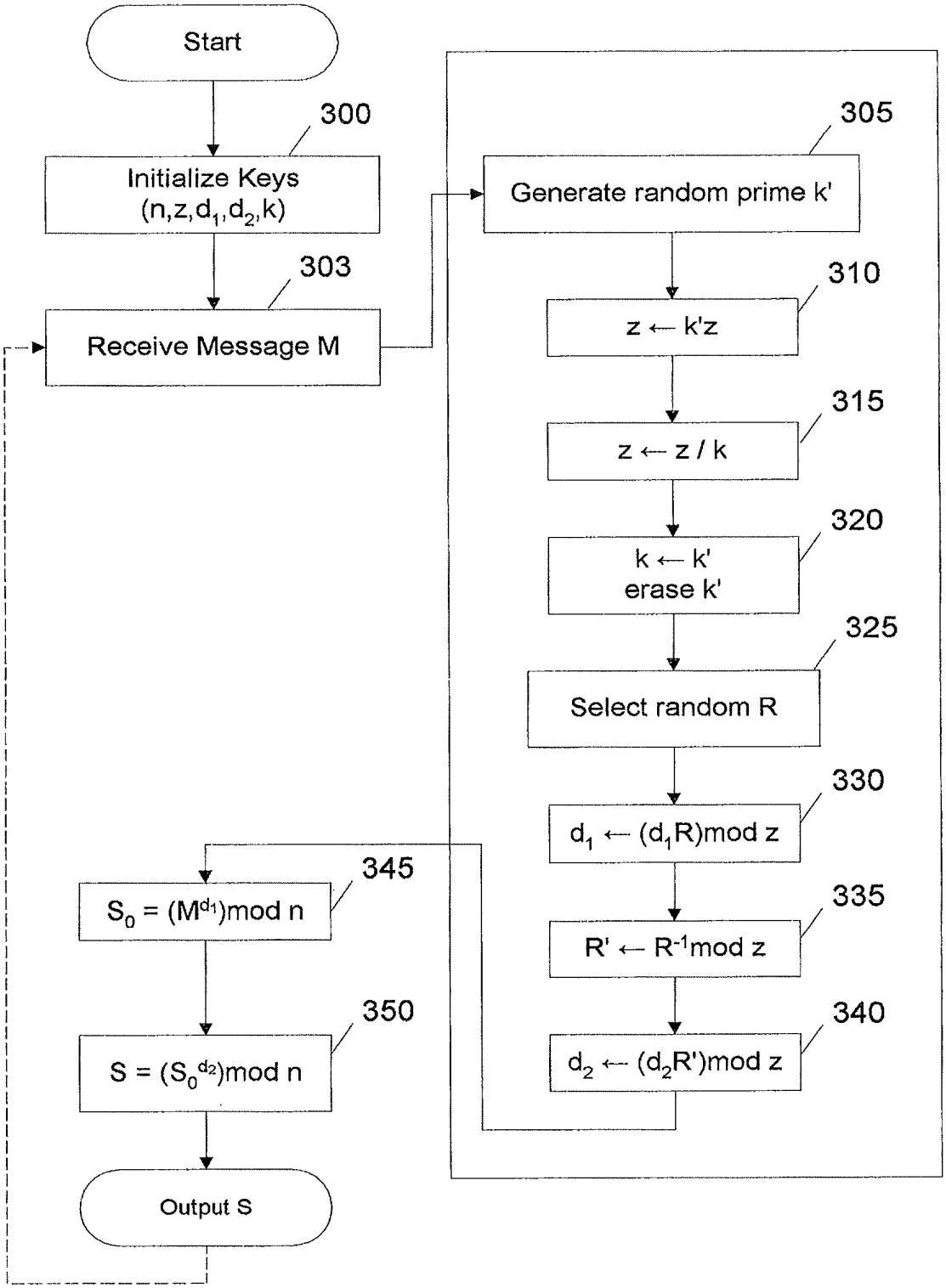
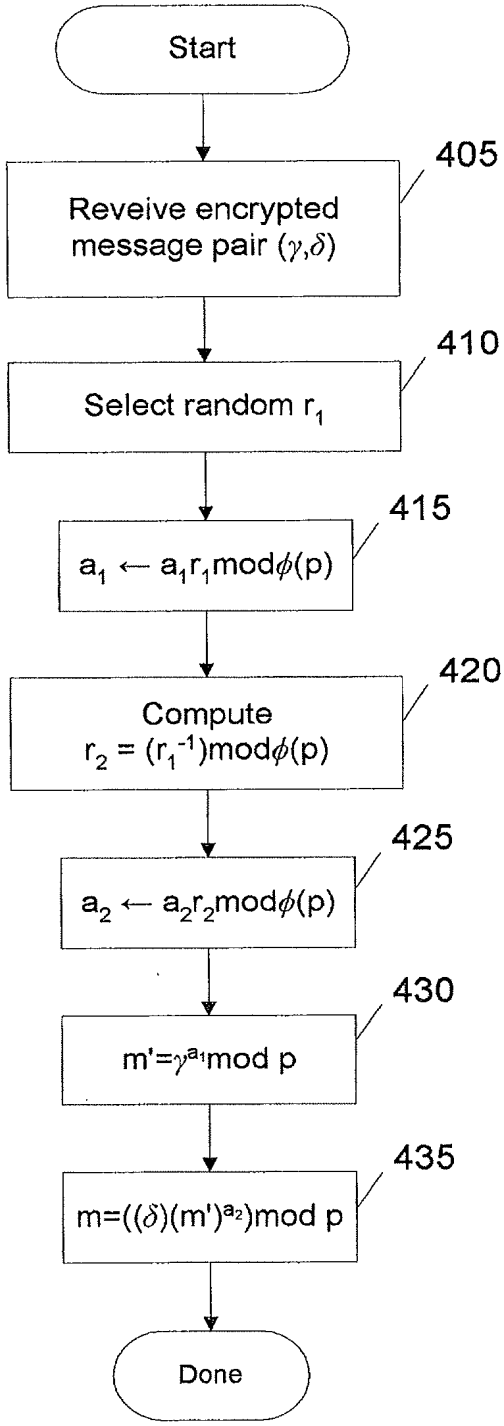

Start
300
Initialize Keys
(n,z,d1,d2,k)
303
Receive Message M
305
Generate random prime k'
310
z ← k'z
315
z ← z / k
320
k ← k'
erase k'
325
Select random R
330
d1 ← (d1R)mod z
335
R' ← R-1mod z
340
d2 ← (d2R')mod z
345
S0 = (Md1)mod n
350
S = (S0d2)mod n
Output S

FIG. 4


Start
Reveive encrypted message pair (γ,δ)
405
Select random r_1
410
$a_1 \leftarrow a_1 r_1 \bmod \phi(p)$
415
Compute $r_2 = (r_1^{-1}) \bmod \phi(p)$
420
$a_2 \leftarrow a_2 r_2 \bmod \phi(p)$
425
$m' = \gamma^{a_1} \bmod p$
430
$m = ((\delta)(m')^{a_2}) \bmod p$
435
Done